Theoretical Computer Science 64 (1989) 175-189
North-Holland

DESIGN FACTORS FOR PARALLEL PROCESSING BENCHMARKS

Gordon LYON

Parallel Processing Group, National Computer Systems Laboratory,
National Institute of Standards and Technology, Gaithersburg, MD 20899, U.S.A.

Abstract. Performance benchmarks should be embedded in comprehensive frameworks that suitably set their context of use. One universal framework appears beyond reach, since distinct architectural clusters are emerging with separate emphases. Large application benchmarks are most successful when they run well on a machine, and thereby demonstrate the economic compatibility of job and architecture. The present value of smaller benchmarks is diagnostic, although sets of them would encourage the parametric study of architectures and applications; an extended example illustrates this last aspect.

1. Introduction

Applications of parallel processing place an emphasis upon performance which differs from that commonly seen for serial processors. The user of a concurrent processor quite often wants his program to run *faster* [1], i.e., he wants an improved turnaround. Emphasis upon a single job running stand-alone means that variance in any general characterization will be higher. However, a program cannot be characterized and its performance predicted with confidence unless it has a close fit to available benchmark models. A benchmark result for some machine is really a statement of confidence in one's understanding of the application features that the benchmark represents and the manner in which it exercises the machine. When this understanding is highly uncertain, a "measurement" ceases to quantify anything. It is this problem, the careful and deliberate framing of benchmark questions, that burdens measurements on parallel processor machines. While the structures for parallel architectures have become more diverse [19], understanding of the various architectural tradeoffs has not maintained pace. In addition, the once popular and relatively easy *empirical* technique of capturing instruction mix streams does not fully characterize salient features of parallel executions. In particular, communication patterns are not evident [6]. This failure of a relatively easy, if not always accurate, method of application characterization has forced a closer analytic examination of applications.

Each application uses parallel architectural features in its own combinations and strengths. When uniprocessors were dominant, the underlying model did not often differ substantially from one machine to another, at least not from the user's perspective. Nonetheless, from a micro-architectural view there has always been a clash between a compiler's ideal target (application virtual instructions) and a

machine's capabilities (real instruction architecture). But parallel processing elevates structural clashings to higher levels. Thus, an application with a detailed set of interacting parallel pieces may or may not be architecturally suited for a certain parallel machine. The clash is larger than instruction mismatches. It includes process patterns-of-fetching from memory, communication modes among independent threads-of-execution, and levels of the communication traffic.

Ideally, one would like benchmarks that are pertinent, accepted, and simple [6]. But at present, accurate prediction from a benchmark set is harder than running actual programs. A predictive mechanism rests upon a thorough understanding of underlying parallel systems and their relation to applications. This understanding is very rough. Consequently, many machine "measurements" are timings of large, poorly characterized calculations that are nearly complete workloads. There is little economy in this, but in many circumstances there are also few alternatives.

1.1. Levels of benchmark focus

An application program runs, computes answers, and (thereby) implicitly represents all the complexity of the problem it addresses. Unfortunately, application programs are often very large, with organizations and detailed performances not easily understood. Minor changes in data may yield large excursions in performance. Typically, an application job becomes an important benchmark because it may compute heavily or epitomize a whole class of problems that are important. It may match a test machine architecture and exercise the machine well. The weaknesses in a large benchmark lie in the cost of getting it onto a machine and in interpreting its results. Outsiders are usually reluctant to invest large blocks of time to learn about other fields of application for ancillary purposes. Consequently, they often cannot readily extrapolate from full-application test results. Indeed, they may not understand enough of a program to vary its input meaningfully. Similarly, the size of a real application may discourage a detailed analysis of its instruction-level demands. Yet this is often necessary to gain insight, especially when low-level demands vary greatly with program parameter sets. Production-sized "codes" demand much from those who use them to investigate computer performance. Certainly, large benchmarks are often instrumental in selecting institutional computers. However, this use is more of a service for an organization's bureaucracy than it is a contribution to understanding parallel computation.

Algorithms in-the-large can be paradigms for solving whole subsystems. Often they are templates which describe, outside-in, how solutions should be constituted. A description consists of outer layers of code, with the variable (substitution) opportunities occurring within. In contrast, programming techniques are algorithms-in-the-small, and are usually inside-out in orientation. A file access method is an example. Here the code is wrapped with the user's application; the view is that of supplying standard code inserts. Neither approach is an entire program; each is an abstracted characteristic of some computation. Both relate to smaller benchmarks in a very important way: they strive to embody computations that are significant

for large numbers of applications. Paradigms and techniques provide a focus for ignoring programming language *as long as it is adequate.* They focus upon mechanistic details that are *truly important,* independent of a given program (*parallel prefix* is an example). Both approach performance from an algorithmic viewpoint of relative magnitudes of improvement rather than simple percentage gains of implementation. While structural soundness (e.g., proof-of-correctness) may be very important, it is secondary to the main argument for each. And both supply practical software design recommendations based upon *measured* performances. This may exclude certain elegant theoretical approaches whose deficiencies are clearly abundant once tried.

Many MIMD systems use process-level parallelism: how and at what costs do various cooperating processes start, run, synchronize and exchange data? A later example explores communication within this context. Generally, the processes themselves can be *synthetic* (need not compute actual answers) and so can be simplified greatly. Some aspects of real interest include synchronization, process creation, memory allocation, context switching, message transmission-reception, shared memory accessing and task scheduling. A specific problem determines which aspects are given emphasis, e.g., in studying communication, process creation may be of little interest.

The instruction level of performance characterization is very important in the design of machines for commercial production. The approach often has been empirical; large, representative workloads are characterized at the instruction stream level. But instruction mix streams have much diminished utility with parallel architecture because data movement is not characterized. Naturally, the *address reference sequence* does help in the exploration of memory-system characteristics, and some of its limitations (extreme sequence lengths) have been eased through recent simplifications that focus upon cache misses; see [24] for an extended discussion. Another approach at the instruction level is the time-stamping of various events in a program. This should be done with little if any overhead [17]. Machine event detail is also critical in verifying that performances characterized at higher levels are accurately based upon actual resource utilizations. For here one can isolate and document the causes of anomalies observed at other benchmark levels. A weakness with many benchmarking "measurements" is that they are based upon a conjectural machine model. As a consequence, they can be inaccurate in predicting changes from a host modification. A model's detail depends heavily upon the target architecture. The shared memory machine will have concerns of processor cache hits, interconnect and memory contention, and load balancing. A message-passing system's principal concern might be transmission latencies, bandwidth, error recovery and flow control.

Machine capabilities should fit together in balanced capacities to define a reasonable performance regime [10]; this also provides a basis for examination. It is possible, of course, that a machine has some clever twist that renders the questions unsuited; in such cases further explanation by the designer is in order (e.g., some

machines can tolerate a narrower usefulness because this makes them significantly cheaper). Details of interest include local and global memory sizes, memory-to-processor bandwidths and significant latencies, processor bandwidths, I/O capabilities, "memory move-around" capability, and processor-to-processor bandwidth and latency. Together these constitute dimensions of elementary capability and balance of which no one feature is more important than another. This balance-of-stress is an ideal which is never fully achieved, but one aim of performance measurement is to identify economical approximations.

2. Frame of reference

The problem of a conceptual framework for benchmarking is well recognized. A good organization of ideas is necessary to handle the vast horde of application programs and the many available machines. NRC [1] has proposed (after ideas of Joanne Martin [23]) five stages in performance evaluation:

(1) determine major application areas and solution techniques,
(2) elect representative programs covering these,
(3) define parameters for models of architecture and the application,
(4) define metrics for environment and performance of the models,
(5) assess relationship between the computational and architectural models.

Of course, this can be read as "Wish upon a star," something considerably easier said than done. The experiments by Weems and his associates on image understanding benchmarks demonstrate the considerable level of effort that a good characterization may demand [25]. (Image understanding is certainly a subculture of the overall parallel community.) Etchells and Nudd [6] remark:

> "... The problem of selecting representative algorithms is itself complicated by the breadth of the field [Image Understanding], the wide range of approaches to any given IU sub-task, and by the fact that there are many areas in which there is no clear consensus as to the best algorithm for performing a given task .... Ideally, what we would like to do is to find the lowest level of program modules with the greatest degree of applicability across the entire range of IU algorithms."

For a highest level of organization, Etchells and Nudd borrow a six-dimension taxonomy. Their approach stresses a range of processing requirements, from those of applications down to concerns appropriate for hardware alone. The proposed levels are consonant with their view:

Applications. This layer defines requirements, and clarifies *need.*

Algorithms. The organization and detail give actual *answers.*

Parallel (process) structure. This is an *abstraction* of parallel computation on process-oriented systems. Finer grained organizations will need the level relocated as appropriate.

Instructions. Sharper detail provides *isolation* of anomalies. Special hardware may be necessary to get convenient, accurate, unperturbing measurements.

Hardware resource utilization. Balance and overall envelope serve to define gross *capacities* and *limitations* of a machine.

2.1. Utility versus benchmark size

There is the issue of where in the layering one should choose a design focus; a choice clearly depends upon need. If the community is an applications group with principal interest apart from parallel computing, then it selects representative programs (designated *large codes*) that demonstrate whether a new parallel machine can service applications. The second view, expounded by Etchells and Nudd, seeks more fundamental building blocks. *Small metrics*, like a test for blood pressure, are narrow indicators that highlight potential threats [22].

Large codes and small metrics have complementary strengths that are instructive. Ineffectiveness of a parallel host can be difficult to interpret with a large code of twenty to forty thousand lines. While the host machine may be incompatible, it can also be that the program is trying to do something in a particularly unsuitable way. Yet the volume of code impedes an easy identification. Similarly, a collection of good scores on small metrics (e.g., the LFK set [15]) indicates that a machine enjoys isolated strengths. But without reliable rules of synthesis for extrapolating the results, predictions for complex computations are uncomfortably loose and conjectural. Table 1 summarizes some differences in utility given two outcomes from running.

Table 1

	Nice speedup	Little improvement
Large code	(+) Indicates application/ architecture compatibility	(−) *Identifying reasons often bothersome, difficult*
Small metric	(−) *Rules of extrapolation not general at this time*	(+) Isolates deficiencies quickly and clearly

The best way to buy a machine is to run a very representative job and have it do well. However, if performance is inferior, then small metrics are far more convenient and specific. The current state of parallel computing is far too chaotic to place great faith in any general synthesis from small metric results, since to do so assumes that a good model exists. As for large codes, the best way to handle complications of any magnitude may be simply to set them aside. This recommendation is especially true for "dusty-deck" codes whose structures antedate most modern software engineering practices.

3. Additional factors in realistic design

The level of design that a benchmark addresses will affect its overall utility to various people. Larger benchmark codes test production capabilities well enough, but the amount of code can be unattractive. The very number of applications renders a full repertoire of larger codes simply hard to attain. Such a collection will lack economy of size and thought, and generate a maintenance burden. A low level

benchmark is simpler to interpret, gives reproducible results and is easy to code: unfortunately, it may say nothing about germane parallel interactions. (E.g., cache hits may be disproportionately high.) As Etchells and Nudd have indicated, some middle ground is often an attractive target.

Membership selection for benchmark sets varies considerably. Nonetheless, two common and diametric poles exert an influence:

(i) the empirical—culling from real application codes those program figurations which arise frequently, and

(ii) the axiomatic—working within known organizational relationships which must be satisfied.

The Livermore FORTRAN Kernels [15] convey a strong feeling of (i), whereas the example in Section 4 and the "Organick Raspberry" [20] are more axiomatic, (ii). Circumstance and choice will always determine some proportional mix of the two. Ideally, one would like to isolate *all* important programming and algorithmic factors and test for them with measurement benchmarks. This is not possible, of course, since a general set will be very large. Eventually, a method of synthesis from important benchmark "bases" may solve some of the size-of-set limitation. A set of metrics would cover all pertinent dimensions, with more complex cases being extrapolations from basis-set figures. However, at this time, accurate predictions of interactions are likely to be more difficult and less accurate than simply writing ad hoc benchmarks to capture the circumstances directly.

Benchmark specification, as with any other design, involves tradeoffs. Certainly, there is considerable latitude in the description of a benchmark. It may be the most simply-stated question, such as, "Is $2^{314959216}-1$ prime?" At the other extreme, it can be as detailed as loadable machine code. The above question on primality is a quite portable benchmark specification, whereas the machine code version is easily run, *provided that a suitable system is under test*; a narrow target of use allows simplifications and assumptions that must otherwise be accounted for. An example of this accounting is the library used with a collection of coded benchmarks. If a benchmark will not always be run on the same system, then arguments can be made that its library should be bound to it. This is commendable, but it means that someone must write and maintain the library; furthermore, its routines will be inferior for some architectures and implementations. To continue, specification can be resolved at several levels. A natural language description (German, French, ...) will provide the most portable, applications-oriented possibility. It is closest to application requirements; it leaves miles of latitude for implementation. Pseudo-code is noticeably more concrete. The opportunity for misinterpretation is much diminished and the portability of the benchmark is not greatly degraded *on machines similar to the virtual machine assumed in the pseudo-code.* Real code is even more pronounced in its immediacy, the principal variable being the quality of compilers and libraries; even here the range of performance may span a whole decimal magnitude. Finally, machine-loadable code removes all compiler uncertainties, but it also precludes opportunities for running on different product lines.

Data and scoring encompass another important benchmark element. Consider first the scoring of a single test. Without standard input data, one is never quite sure that a result is comparable; many algorithms exhibit wide changes in their execution from what seem to be minor parameter variations. So all testing with a benchmark should ask for comparable computations whenever comparisons are to be made. Specification of output is not quite so crucial because the values are more easily read at face value. Comparabilities are more readily established; 1.55556 may be close enough to 1.6 for a known test. However, if fixed output tolerances are not acceptable, then a detailed "question and answer sheet" must be prepared for scoring each benchmark. And every scoring and evaluation has a scope, the parametric range for a metric set being quite dependent upon what purpose the results serve. For example, when checking a machine for service suitability, the application determines the range. However, this may not "stress" the machine in a manner interesting to an architect trying to explore its regimes of performance. From his viewpoint, which is naturally more bottom-up, the parameter ranges should uncover interesting and significant transitions in execution performance. Without points of transition, he has little idea whether there is a balanced overall capacity for market applications.

Once all single scores are complete, an overall assessment may be needed. This aggregation of individual scores depends upon the evaluation model, which explains why an unqualified overall numeric score can be so misleading. Some evaluations, such as for export control, might benefit from avoiding a single score entirely [2]. And, even when an intuitive consensus exists for an average scoring, it may not be logically correct. (See [4] for passing remarks on arithmetic and harmonic means.) However, the problem of fit is hardly unique to computer evaluation; in experimental statistics, an assumed distribution for a means-calculation, whether geometric, normal or otherwise, is pivotal.

Overall interpretations of a benchmark score are sometimes hard to extract. This is one reason that a fuller framework has been proposed. It forces the user either tacitly to agree with the framework setting, or to supply arguments on his reinterpretation of a benchmark's significance. In either case, the results are certainly much more valuable when explained, so that provision should be made for interpretations of all scores. Otherwise laymen and rascals will draw their own conclusions, and these may *occasionally* be correct! The report on the LFK set is an excellent example of interpretation, although it does follow introduction of the set by roughly seventeen years.

4. An example: metrics for process communication

Process communication is absolutely necessary in parallel processing. Without mutual exchanges of data, cooperative computations are precluded. A system's capabilities in communication are pivotal to its algorithm designs and techniques

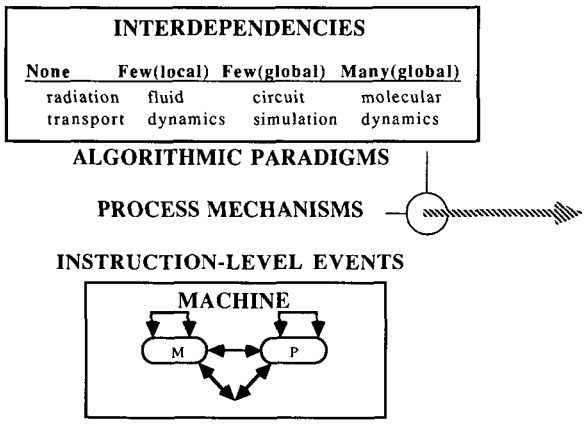
of programming. *Process communication granularity* denotes herein an informal indication of size or resolution. (See [12] for a more ambitious attack on the problem of computational grain.) Thus a fast synchronization method is fine-grained, and a slower one, say five-second polling, coarse. Similarly, fine-grained transmissions have but a few bytes, whereas coarse messages are long. Granularity is important because it relates both to the cost of writing software and to the cost of the underlying hardware. It always implies a certain minimal amount of useful computation be associated with each transaction so that overheads are rendered insignificant. Generally, coarse-grained MIMD hardware is less expensive, but it requires coarse-grained algorithms, which can be more difficult.

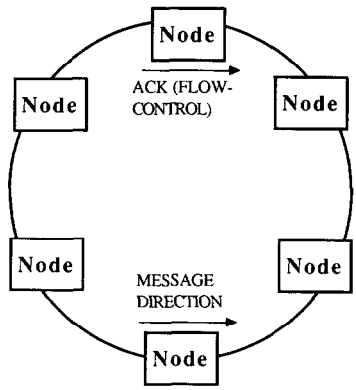
The investigations pursue process communication granularity within and across various architectures, although shared memory architecture has been given greatest initial emphasis. *Shared memory* denotes storage mutually accessible by all processors at approximately the same cost; it supports communication *by-reference*, i.e., through pointers. *Message passing* systems have memory attached privately to each processor, so that processors communicate only through explicit transmissions of whole messages. From a programming view, messages are a *by-value* form of communication. Accommodating both communication styles within the same test layout requires advance planning. The code for message passing is designed first. Translation—in an informal sense—is then made to a shared memory version. This direction of designing *from* message passing *to* shared memory is very important. It can be quite difficult to cast some shared memory codes into a suitable message passing equivalent [9].

A two-dimensional general framework (see Fig. 1) has been chosen to emphasize communication. The abscissa delineates several broad application modes by communication dependencies; computational objects that can be scheduled independently (à la radiation transport), locally-dependent calculations (fluids), scattered global dependencies (circuit simulation), and many interdependent global calculations (molecular dynamics). While these modes are vast simplifications, they are a serviceable organization. The ordinate depicts a degree of abstraction away from the physical machine, as discussed earlier. At the highest level are the applications, classified by their component dependencies. This top level is primarily a requirements specification. Next is an algorithmic layer which establishes the functionality of the *serial process code* and yields the application's detailed communication requirements. Lower yet is a (parallel) process level in which computation loads, communication patterns, and transmission protocols dominate. And beneath this, instruction or machine event measurements should ensure that what is seen with software benchmarks is a true pattern of machine utilization. At the bottom there is a machine that should be well-balanced.

4.1. A ring

For each communication dependency (x-axis) at the process mechanism level (y-axis), one can define a logical (but hardly unique) communications structure.





Figs. 1 and 2. (1) Framework, (2) ring pipeline.

The role of the logical structure is to provide an abstract model of communication divorced from fine details of

(i) the original problem, or

(ii) algorithmic and coding features not related to process communication. (Communication denotes both synchronization and data transmission.)

Certainly there may be numerous acceptable models for a given application. However, to study process communications, specific examples must be chosen and tested; a ring structure has been selected to exemplify global process dependencies. It is implemented with parameter variations as follows:

(A) synchronization (busy-wait; polling; interrupts),

(B) mode of transmission (by-value; by-reference),

(C) message length (short to long).

Further variation is necessary within gross parameter selections. For example, polling introduces the notion of *frequency* which must be explored. Computation per datum should be adjustable. In addition, the variance of processing each datum can be set by another parameter.

The synthetic ring benchmark (Fig. 2) for global dependencies works as follows: each of n nodes will originate x messages, and additionally, process all other messages passing by. The number x of messages and their length y are parameters. Each message travels around the ring while being "processed" synthetically by each node. A message that returns to its origination node is removed from the ring traffic. A new message is sent unless all x have been sent. When all nodes have sent and received all of their messages, the ring of processes is dissolved and the results are reported. Communication is asynchronous, with message traffic regulated by a simple form of flow control. This keeps slower nodes from being overrun with messages. Such control is essential on systems that cannot control buffer overflows. In this

preliminary study, messages are acknowledged on a one-to-one basis. Thus, at most, a process (node) will have one waiting message.

The ring executes in $\Theta(n^2)$ on a serial machine. This can be seen by doubling the ring size: each message goes twice as far and there are twice as many messages. Now, an architecture such as the hypercube can assign a physical processor to each ring node (process). In such cases, one expects running in $\Theta(n)$ (based upon the number of messages each node will see). Observation bears this out: hypercube performance adheres closely to a straight line, whereas a processor-limited shared memory machine begins to slow in a nonlinear fashion. For p processors in a shared memory machine, the deviation from linear appears as the ring size reaches p nodes (assuming that interconnect and memory remain unsaturated, but that computation is heavy). Prior to this, each ring node has a processor and the system still has a processor to do its chores. At $n = p$ the ring needs all p processors, but the system also needs one occasionally. Hence time-to-complete begins to take longer than linear, and each additional virtual ring node merely worsens processor contention.

There are three sets of illustrative ring data. Discussion centers upon a shared memory machine with six processors unless noted otherwise. The plots include

(1) handling very short messages,

(2) interrupts, polling, and busy-waiting on long messages, and

(3) influences from polling frequency.

No claim is made that these sets are fundamental to all or any other parallel programming applications. Yet, the results do demonstrate that even a simple benchmark can extract revealing performances. Thus, it and other similar metrics may be quite useful in quickly characterizing salient parallel performance features. Good characterizations can only shorten the task of designing good programs for a new machine.

Experiments were conducted sending the same amount of information throughout the ring in varying degrees of "chunking." Longer messages were fewer in number. The times-to-complete in Fig. 3 are plotted (log–log scales) against the message lengths. The overall constraint is $(\textit{message length}) * (\textit{number of messages}) = \textit{CONSTANT}$. The results in Fig. 3 are for shared memory, by-reference transmission; essentially, a pointer is passed around. As to be expected, whenever there is a free processor to assign to each node process in the ring, busy-waiting works very well. In Fig. 3 one sees numerous time-to-completion curves for rings of two to seven nodes run on a six-processor shared memory machine. For six or fewer nodes, variation in message length (and by constraint, number of messages) is not nearly as critical for busy-waiting as it is for interrupts: busy-waiting does not incur the latencies of context-switching and system service that interrupts and polling entail. Very fine grained communication is feasible. But busy-waiting squanders processor capacity, and one expects that once processes exceed processors in number, busy-waiting will be definitely inferior. Indeed, the line BW(7) in Fig. 3 (a seven node ring) depicts a much worse performance than that for interrupts, INT(7). The ring routine can do by-value message transmissions as well as by-reference, but, for very

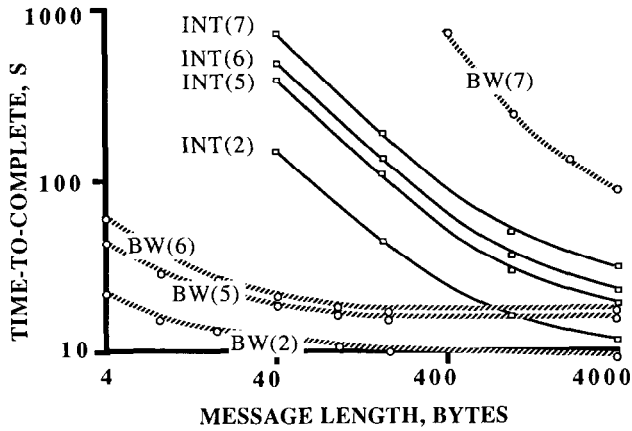


Fig. 3. Time vs. message lengths.

short messages and shared memory this distinction is not crucial. However, Fig. 4 depicts the performance of an older by-value (message-passing) architecture in which the system's frame size for messages was fairly long. Any attempt to send short messages mostly swamped the communications network: Whole message frames were sent no matter how few bytes were used. Whenever the shortest transmissions were attempted, the system would crash from communications overload. The sawtooth pattern of Fig. 4 reflects the influence of the message frame.

Drawing from the above, one can conclude that the ring benchmark is useful in establishing quickly which message lengths need extra concern in designing algorithms for a system. The ring also tests the robustness of system communications.

4.2. Interrupts, busy-waits, and pollings

The next variations examine several implementations of synchronization. To simplify, messages are very long (8 kilobytes). This removes many problems with

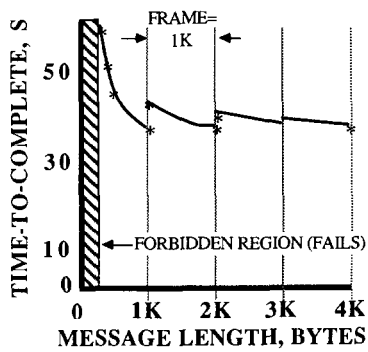


Fig. 4. BW(2) stress on hypercube variant.

short messages. Generally for the ring, interrupts are the most practical. That is, for larger, computationally heavier applications with medium to long messages (both perhaps common), interrupts give the fastest executions with the least trouble. Their primary message grain limitation arises from system overheads; interrupt handling requires operating system service, and this precludes shorter messages. When messages are longer, as in Fig. 5, interrupts outperform busy-waiting and are not nearly as sensitive to tuning as polling. Fig. 5 also displays time-to-completion curves for *by-value* modes of transmission. This corresponds to message-passing, since no permanent storage of a message is allowed in shared memory. A message must be copied totally into a processor's private memory area, processed, and copied back to shared memory for transmission. While this mode is considerably more secure, it is understandably slower for longer messages. Milde et al. [16] remark on their experiences with the Aachen M^5PS cluster machine, "The comparison reveals a significant overhead of transparent message passing as against synchronized communication via shared variables." The point is worth noting because shared memory performance declines considerably when using a *by-value* paradigm.

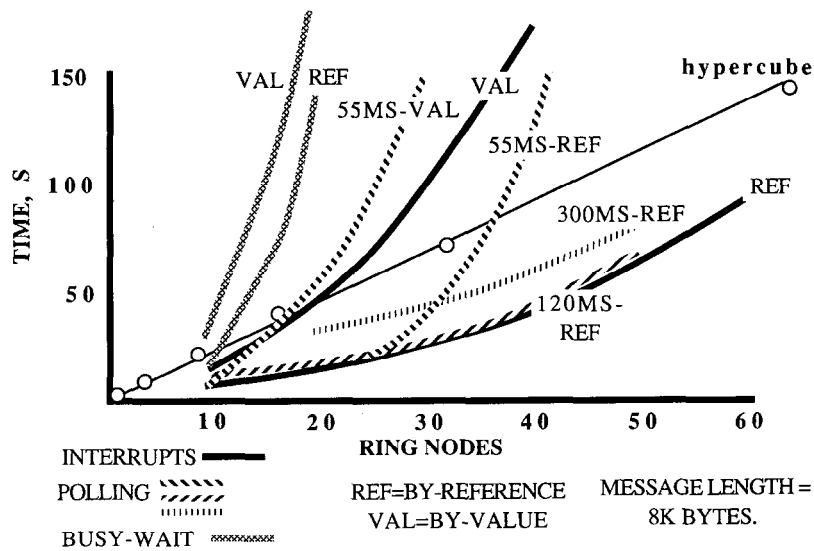


Fig. 5. By-value and by-reference for three methods.

For larger applications with medium to long messages, polling offers performances ranging from excellent to inferior (Fig. 5). The sensitive nature of the ring benchmark regarding polling is clear from the abrupt transitions of the curves in Fig. 6, which depicts *relative* time-to-completion against polling frequency in milliseconds. While a good polling frequency for the ring structure is easy to identify, the curve changes with problem parameters. Thus, each distinct set of problem parameters may need a new adjustment. This tuning can be amazingly delicate, with a millisecond change

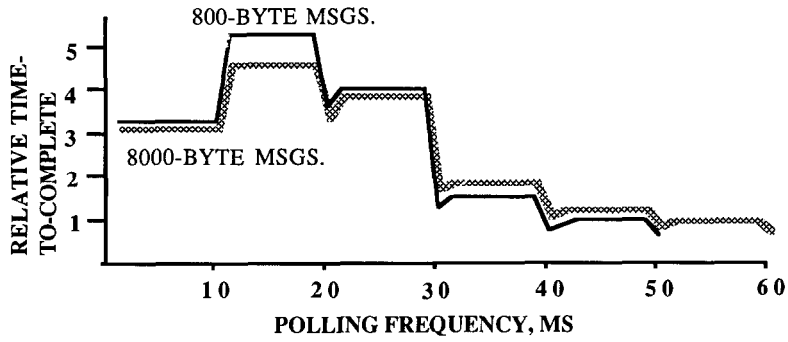


Fig. 6. 30 Node ring ω/polling on some systems.

in polling frequency resulting in a nearly three-fold increase in time-to-complete. As the ring grows larger, it amplifies performance differences that result from polling frequency. Thus 20 ms polling may run ten percent faster than 19 ms polling for a ring of ten nodes but yield a factor of two or three times faster for a ring of thirty nodes. Fig. 6 shows two runs, each normalized *within their own time-scale* for time-to-completion, plotted against polling frequency in milliseconds. The obvious "steps" are at system clock "tick" multiples. Polling at more than one but less than three clock "ticks" seems an especially poor choice on the specific shared memory system. (Among three other vendors, two show a similar polling effect.) Initial instruction level measurements on the polling request show a very high variance in its service times. Locking processes onto processors improves the situation considerably, and points suspicion at the process scheduler.

5. Conclusions

The structured framework and the ring benchmark provide one benchmarking approach. Although more experience is needed both running the ring on various machines and with other related benchmarks, the framework appears promising as a structure for organizing debate on communication aspects.

In general, large codes and small metrics can take valuable roles that complement each other. Codes fulfill a present need to assess the utility of parallel processing for real applications. They assure purchasers of radical architectures that, while the full envelope of performance is not understood, their new machine does demonstrate powers that are judged an advance over services otherwise available. In contrast, metrics reflect a piecemeal basic understanding. A metric is an element of a model, a simplification along a dimension of interest. Small metrics should promote easy parametric studies that isolate system anomalies and encourage directions of enhancement.

Overall, discussion has covered roles, settings, and summaries for benchmark sets. Benchmarks reflect the field of parallel processing itself; as it becomes more orderly, problems in performance characterization will also ease. There are several recommendations:

(i) Performance metrics should be embedded in more comprehensive frameworks that can set the context of the experiments.

(ii) One universal framework is beyond reach, since distinct clusters of use are emerging with separate emphases.

(iii) Large application benchmarks are most successful when they run well, thereby demonstrating economic compatibility of job and machine.

(iv) For now, smaller metrics are more diagnostic and preventive than predictive.

Acknowledgment

Thanks go to R. Carpenter, S. Lakshmivarahan, C. Lyon and R. Snelick for their observations and comments. The Advanced Computing Research Facility, Argonne National Laboratory, Argonne, IL, and The Supercomputing Research Center, Lanham, MD., kindly provided systems for some measurements. This text is abridged from an earlier National Bureau of Standards Internal Report NBSIR 87-3580, and the work is partially sponsored by the Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, VA 22209, U.S.A. under ARPA Order No. 7223, April 15, 1987.

No recommendation or endorsement, express or otherwise, is given by the National Institute of Standards and Technology or any sponsor for illustrative commercial items in the text. Contributions of NIST have no U.S. copyright.

References

[1] An agenda for improved evaluation of supercomputer performance, Report prepared by the Committee on Supercomputer Performance and Development, National Research Council (Washington, DC, 1986).

[2] D. Bailey, E. Brooks, J. Dongarra, A. Hayes, M. Heath and G. Lyon, Benchmarks to supplant export "FPDR" calculations, NBSIR88-3795, June, 1988.

[3] R.J. Carpenter, Proposed computer performance measurement hardware, Parallel Processing Lab. Note No. 20, ICST, NBS, March, 1986.

[4] J. Dongarra, J.L. Martin and J. Worlton, Computer benchmarking: paths and pitfalls, *IEEE Spectrum* **7** (1987) 38-43.

[5] J. Efron, Computers and theory of statistics: thinking the unthinkable, *SIAM Review* **21** (1978) 460-480.

[6] R.D. Etchells and G.R. Nudd, Software metrics for performance analysis of parallel hardware, Hughes Research Laboratory (Malibu, CA) Report, circa 1983, under Darpa Order No. 3119; Reported at *Alvey-DARPA Workshop on Benchmarking Parallel Architectures*, London, October 10-11, 1984.

[7] D. Ferrari, Considerations on the insularity of performance evaluation, *IEEE Trans. on Software Eng.* **SE-12** (1986) 678-683.

[8] J. Gait, A probe effect on concurrent programs, *SOFTWARE—Practice and Experience* **16** (1986) 225-233.

[9] T. Hikita and K. Ishihata, A method of program transformation between variable sharing and message passing, *Software—Practice and Experience* **15**(7) (1985) 677-692.

[10] D. Hillis, Remarks on parallel architectural similarity, in: *Proc. 12th Annual Int. Symp. on Computer Architecture*, Boston, June 1985.

[11] J.E. Huss and J.A. Pennline, A comparison of five benchmarks, NASA Technical Memorandum 88956, February, 1987.

[12] C.P. Kruskal and C.H. Smith, On the notion of granularity, *J. Supercomputing* **1** (1988) 395-408.

[13] G. Lyon, A fast, message-based, tagless marking, in: M. Heath, ed., *Hypercube Multiprocessors 1987: Second Hypercube Multiprocessor Conference*, Knoxville, September 1986 (SIAM, Philadelphia, 1987) 78-82.

[14] G. Lyon, On parallel processing benchmarks, NBSIR87-3580, June, 1987; available through National Technical Information Service (NTIS), Springfield, VA 22161, U.S.A.

[15] F.H. McMahon, The Livermore FORTRAN kernels: a computer test of the numerical performance range, Report UCRL-53745, December 1986, Lawrence Livermore National Laboratory; available through National Technical Information Service (NTIS), Springfield, VA 22161, U.S.A.

[16] J. Milde, T. Plueckebaum and W. Ameling, Synchronous communication of cooperating processes in the M^{5}SP multiprocessor, in: *Proc., CONPAR 86*, Lecture Notes in Computer Science **237** (Springer-Verlag, New York, 1986) 142-148.

[17] A. Mink et al., Simple multiprocessor measurement techniques and preliminary measurements using them, Parallel Processing Lab. Note No. 23, ICST, NBS, March 1986.

[18] M.G. Natrella, *Experimental Statistics* NBS Handbook **91** (August 1963).

[19] S.W. O'Malley and J.B. Gilmer Jr, Survey of high performance parallel atchitectures, Report BDM/ROS-85-0988-TR, BDM Corp., Dec. 1985.

[20] E. Organick and R. Asbury, The iPSC stress tester (Organick Raspberry), in: M. Heath, ed., *Hypercube Multiprocessors 1986: First Hypercube Multiprocessor Conference*, Knoxville, August 1985 (SIAM, Philadelphia, 1986) 38-42.

[21] C. Pavelin, *Notes on Alvey-DARPA Workshop on Benchmarking Parallel Architectures*, London, October 10-11, 1984 (Rutherford Appleton Laboratory, Chilton, 1984).

[22] J.B.G. Roberts, Evaluating parallel processors for real-time applications, in: *Proc. Conf. on Vector and Parallel Processors in Computational Science III*, Univ. of Liverpool, August 1987; also *Parallel Computing* **8** (1988) 245-254.

[23] S.B. Salazar and C.H. Smith, *National Bureau of Standards Workshop on Performance Evaluation of Parallel Computers*, NBSIR86-3395, July 1986; available through National Technical Information Service (NTIS), Springfield, VA 22161, U.S.A.

[24] H. Stone, *High-Performance Computer Architecture* (Addison-Wesley, New York, 1987).

[25] C. Weems, A. Hanson, E. Riseman and A. Rosenfeld, An integrated image understanding benchmark: Recognition of a $2\frac{1}{2}$D "Mobile," COINS Dept., Univ. of Massachusetts, circa May 1988.